CHAPTER 11
MOVING A LOT OF DATA

Most of the examples covered so far have been moving small amounts of intermittent data using the build-in HID device driver. This approach has the benefit of being simple to implement since the operating system contains all of the driver code necessary to enable an application program to monitor and control the device. An overview of the "Buttons and Lights" solution of Chapter 7 is shown in Figure 11-1. We wrote the PC host application program and all of the I/O device firmware to implement this solution.

<<insert diagram here>>

Figure 11-1 Overview of HID application

Many applications require a large amount of data to be moved and, conceptually, the solution shown in Figure 11-2, is the same as the HID application. Unfortunately the Windows operating system does not have a simple BlockIO transfer driver so there will be additional steps to be completed in this design example. I don't like changing too many variables at the same time so I shall use the same hardware as the HID example – the key points of a BlockIO transfer example will be covered in depth so that you can understand how to implement your specific design. Most of the firmware developed in Chapter 7 will be reusable so I will only focus on the differences in this chapter.

<<insert diagram here>>

Figure 11-2. Overview of a BlockIO transfer application

Most BlockIO devices operate on discrete elements of data such as a single page or a single image and some method of separating multiple blocks is required. Typically an I/O device will operate on a single block at a time and will indicate when the next block can be transferred. I chose to use a HID control/status channel for this "separating" function since we know exactly how this works and it neatly partitions the control/status and data transfer functions into two well-defined interfaces.

Step 1: Design the Hardware

A real BlockIO device would spend a great deal of time in this step. A printer, for example, would receive a page of data from the USB then would have to drive motors to move paper and a print head and would have to put ink onto the paper. This book assumes that you already know how to do this – you design I/O equipment for a living but need to understand how to add the "USB ingredient".

The focus of this design is "moving a lot of data" so this example receives blocks of data as fast as possible and stores it in its local memory. Once all of the blocks are received (up to 32K bytes in this design example) an operation is performed to change the data in some way and it is then made available for a "bulk read".

This example will use the same hardware as the "buttons and lights" example since the USBSIMM contains a convenient 32K buffer.

Step 2: Complete the Descriptors

From the descriptor's perspective, adding a bulk in and a bulk out endpoint is a small addition endpoint to the descriptor defined in Chapter 7. Figure 11-3 shows the descriptor set for our "BlockIO" example.

```
; This example has one Device Descriptor with One Configuration and Two Interfaces:
;        HID:      One HID Descriptor - to make PC host software simpler
;                  One Endpoint Descriptor - for HID Input Reports
;                  One Report Descriptor - one byte IN and one byte OUT reports
;        BlockIO:  Two Endpoint Descriptors
         CSEG
DeviceDescriptor:
         DB       18, 1             ; Length, Type
         DB       10H, 1            ; USB Rev 1.1 (=0110H, low=10H, High=01H)
         DB       0, 0, 0           ; Class, Subclass and Protocol
         DB       EP0Size
         DB       42H, 42H, 20H, 42H, 0, 1; Vendor ID, Product ID and Version
         DB       1, 2, 0           ; Manufacturer, Product & Serial# Names
         DB       1                 ; #Configs
ConfigurationDescriptor:
         DB       9, 2              ; Length, Type
         DB       LOW(ConfigLength), HIGH(ConfigLength)
         DB       2, 1, 3           ; #Interfaces, Configuration#, Config. Name
         DB       10000000b         ; Attributes = Bus Powered
         DB       250               ; Max. Power is 250x2 = 500mA
HIDInterfaceDescriptor:
         DB       9, 4              ; Length, Type
         DB       0, 0, 1           ; ID, No alternate setting, HID uses EP1 In
         DB       3                 ; Class = Human Interface Device
         DB       0, 0              ; Subclass and Protocol
         DB       4                 ; Interface Name
HIDDescriptor:
         DB       9, 21H            ; Length, Type
         DB       0, 1              ; HID Class Specification compliance
         DB       0                 ; Country localization (=none)
         DB       1                 ; Number of descriptors to follow
         DB       22H               ; And it's a Report descriptor
```

```
        DB      LOW(ReportLength), HIGH(ReportLength)
HIDEndpointDescriptor:
        DB      7, 5              ; Length, Type
        DB      10000001b         ; Address = IN 1
        DB      00000011b         ; Interrupt
        DB      EP0Size, 0        ; Maximum packet size (this example only uses 1)
        DB      10                ; Poll every 10msec seconds
BlockIOInterfaceDescriptor:
        DB      9, 4              ; Length, Type
        DB      1, 0, 2           ; ID, No alternate setting, BlockIO uses EP2 In/Out
        DB      0FFH              ; Class = Vendor Defined (needs BlockIO.sys)
        DB      0, 0              ; Subclass and Protocol
        DB      5                 ; Interface Name
BlockIOInEndpointDescriptor:
        DB      7, 5              ; Length, Type
        DB      10000010b         ; Address = IN 2
        DB      00000010b         ; Bulk
        DB      EP0Size, 0        ; Maximum packet size
        DB      0                 ; Ignorred
BlockIOOutEndpointDescriptor:
        DB      7, 5              ; Length, Type
        DB      00000010b         ; Address = OUT 2
        DB      00000010b         ; Bulk
        DB      EP0Size, 0        ; Maximum packet size
        DB      0                 ; Ignorred
ConfigLength    EQU $ - ConfigurationDescriptor
ReportDescriptor:         ; Generated with HID Tool, copied to here
        DB      6, 0, 0FFH        ; Usage_Page (Vendor Defined)
        DB      9, 1              ; Usage (I/O Device)
        DB      0A1H, 1           ; Collection (Application)
        DB      19H, 1            ;   Usage_Minimum (User Defined)
        DB      29H, 8            ;   Usage_Maximum (User Defined)
        DB      15H, 80H          ;   Logical_Minimum (-128)
        DB      25H, 7FH          ;   Logical_Maximum (127)
        DB      75H, 8            ;   Report_Size (8)
        DB      95H, 1            ;   Report_Count (1)
        DB      81H, 2            ;   Input (Data,Var,Abs)
        DB      19H, 1            ;   Usage_Minimum (User Defined)
        DB      29H, 8            ;   Usage_Maximum (User Defined)
        DB      95H, 3            ;   Report_Count (3)
        DB      91H, 2            ;   Output (Data,Var,Abs)
        DB      0C0H              ; End_Collection
ReportLength    EQU $-ReportDescriptor
String0:                          ; Declare the UNICODE strings
        DB      4, 3, 9, 4        ; Only English language strings supported
String1:                          ; Manufacturer
        DB      (String2-String1),3            ; Length, Type
        DB      "U",0,"S",0,"B",0," ",0,"D",0,"e",0,"s",0,"i",0,"g",0,"n",0," ", 0
        DB      "B",0,"y",0," ",0,"E",0,"x",0,"a",0,"m",0,"p",0,"l",0,"e",0
String2:                          ; Product Name
        DB      (String3-String2),3
        DB      "E",0,"C",0,"H",0,"O",0,"1",0
String3:                          ; Configuration Name
        DB      (String4-String3),3
        DB      "H",0,"I",0,"D",0,"w",0,"i",0,"t",0,"h",0
        DB      "B",0,"l",0,"o",0,"c",0,"k",0,"I",0,"O",0
String4:                          ; Interface 1 Name
        DB      (String5-String4),3
        DB      "H",0,"I",0,"D",0
String5:                          ; Interface 2 Name
        DB      (EndOfDescriptors-String5),3
        DB      "B",0,"l",0,"o",0,"c",0,"k",0,"I",0,"O",0
EndOfDescriptors:
        DB      0                 ; Backstop for String Descriptors
```

Figure 11-3. Descriptors for HID+BlockIO example

Note that this single device has two interface descriptors: one defines the HID interface and, during enumeration, the HIDUSB.SYS driver will be matched to this interface; the second interface descriptor defines the BlockIO interface and this will be matched up to BLOCKIO.SYS (described later in the chapter). There are two device drivers to support the two interfaces within this example.

Step 3: Implement Microcontroller Code

Almost all of the firmware created for the microcontroller in Chapter 7 is reusable in this BlockIO example. There are a few changes to make to the Device and String Descriptors to identify a different device and changes to how the Reports are handled but the main addition is software routine to support the added bulk input and bulk output endpoints.

The EZ-USB component, like other higher end USB microcontrollers, supports double buffering on bulk endpoints. This means that there are two physical buffers assigned to an endpoint so the one can be filled from USB while the other is being emptied by the USB microcontroller. This technique improves the throughput of the data movement since it results in fewer NAKs from the USB microcontroller.

When the PC host is sending bulk data to the I/O device we should do everything possible to absorb this data at the maximum rate. It is technically possible for the PC host to send nineteen 64-byte bulk packets in a single, full-speed frame. If the PC host sends us a packet and we are not able to receive it (due to no buffer space available) then we will NAK this transaction. The PC host now has to send the data again – this is a waste of USB bandwidth. A PING-NYET protocol was added for more efficient bus utilization of a high-speed bus – a PING is sent by the PC host to inquire if the I/O device has a buffer available for data; this is a token-only packet. If the I/O device replies with a NAK then the PC host will start another PING cycle. If the I/O device replies with an ACK then the PC host will start on OUT data transfer.

Our full-speed example will service the OUT requests as fast as possible to minimize the NAKs generated. I decided to poll the OUT endpoint in the main routine since this was faster than an interrupt service routine. The EZ-USB must programmatically empty the OUT FIFO and save this data in external memory. There is too much overhead in doing a move-byte in a loop so the routine partially un-rolls the loop with 16 in-line MOVX instructions. The EZ-USB's autopointer is used to speed the loop. I measured up to 13 or 14 packets transferred in a single frame (with 3 or 4 NAKs) so there is still a little room for improvement. Figure 11-4 shows the new MAIN module and the full source for all of the firmware is included on the CDROM.

```
MAIN:
                ACALL    DoTaskList           ; BlockIO is polled for performance
                JMP  MAIN
EnableDevice:
                RET
DisableDevice:
                MOV  B, #0            ; Stop the ECHO function
                RET
DoTaskList:
                MOV  A, B             ; Get task list (8 tasks)
                JB   ACC.0, TASK0     ; Receive BlockIO from PC Host
                JB   ACC.1, TASK1     ; Prepare BlockIO for PC Host
                RET
TASK0:
; PC Host is about to send a BlockIO buffer.  Get ready
                MOV  DPTR, #DataBuffer
                CLR  A
                MOV  PacketCounter, A
                MOV  Timer0LOW, A     ; Clear Timer 0
                MOV  Timer0HIGH, A
                MOV  R1, #LOW(EP2OutStatus)
WaitForPacket:
                MOVX A, @R1
                JB   ACC.1, WaitForPacket     ; Wait for Out2BSY = 0
                SETB Timer0_Enable            ; Time the data transfer
; Receive the packet as fast as possible
; First, set up some loop variables
                MOV  R0, #LOW(AutoPtrHIGH)    ; Use AutoPtr to speed execution
                MOV  A, #HIGH(EP2OutBuffer)   ; Overflows every time!
                MOVX @R0, A
                INC  R0               ; Point to AutoPtrLOW
                MOV  A, #LOW(EP2OutBuffer)
                MOVX @R0, A
                INC  R0               ; Point to AutoPtr.Data
                MOV  R7, #4           ; Assume a full buffer (4*16=64)
EP2OutLoop:
                MOVX A, @R0           ; Get new data. This also bumps ptr
                MOVX @DPTR, A         ; Save new data
                INC  DPTR             ; Unroll loop to speed execution time
                MOVX A, @R0           ; Unroll loop 2
                MOVX @DPTR, A
                INC  DPTR
                MOVX A, @R0           ; Unroll loop 3
                MOVX @DPTR, A
                INC  DPTR
                MOVX A, @R0           ; Unroll loop 4
                MOVX @DPTR, A
                INC  DPTR
                MOVX A, @R0           ; Unroll loop 5
                MOVX @DPTR, A
                INC  DPTR
                MOVX A, @R0           ; Unroll loop 6
                MOVX @DPTR, A
                INC  DPTR
                MOVX A, @R0           ; Unroll loop 7
                MOVX @DPTR, A
                INC  DPTR
                MOVX A, @R0           ; Unroll loop 8
                MOVX @DPTR, A
                INC  DPTR
                MOVX A, @R0           ; Unroll loop 9
                MOVX @DPTR, A
                INC  DPTR
                MOVX A, @R0           ; Unroll loop 10
                MOVX @DPTR, A
                INC  DPTR
                MOVX A, @R0           ; Unroll loop 11
                MOVX @DPTR, A
                INC  DPTR
                MOVX A, @R0           ; Unroll loop 12
                MOVX @DPTR, A
                INC  DPTR
```

```
                MOVX A, @R0           ; Unroll loop 13
                MOVX @DPTR, A
                INC  DPTR
                MOVX A, @R0           ; Unroll loop 14
                MOVX @DPTR, A
                INC  DPTR
                MOVX A, @R0           ; Unroll loop 15
                MOVX @DPTR, A
                INC  DPTR
                MOVX A, @R0           ; Unroll loop 16
                MOVX @DPTR, A
                INC  DPTR
                DJNZ R7, EP2OutLoop
                INC  PacketCounter
                MOV  R0, #LOW(EP2OutByteCount)
                MOVX A, @R0           ; Get actual Byte Length
                MOVX @R0, A           ; Tell SIE we're done with the buffer
                CJNE A, #64, LastOne
                JMP  WaitForPacket    ; Get the next one
LastOne:
                CLR  Timer0_Enable    ; Stop the timer
                MOV  LastPacketValidBytes, A
                CLR  B.0              ; We're done with Task0
                RET
Task1:                                ; Prepare a BlockIO buffer for PC Host
                MOV  DPTR, #DataBuffer
                CLR  A
                MOV  Timer0LOW, A     ; Clear Timer 0
                MOV  Timer0HIGH, A
                MOV  R0, #LOW(AutoPtrHIGH)
                MOV  A, #HIGH(EP2InBuffer)
                MOVX @R0, A
                MOV  R1, #LOW(EP2InStatus)
WaitForBuffer:  MOVX A, @R1
                JB   ACC.1, WaitForBuffer
                SETB Timer0_Enable    ; Time this data xfer operation
                MOV  R0, #LOW(AutoPtrLOW)
                MOV  A, #LOW(EP2InBuffer)
                MOVX @R0, A
                INC  R0               ; Point to AutoPtr.Data
                MOV  R7, #4           ; Assume a full buffer (4*16=64)
EP2InLoop:
                MOVX A, @DPTR         ; Get data
                INC  DPTR
                MOVX @R0, A           ; Fill buffer. This also bumps ptr
                MOVX A, @DPTR         ; Unroll loop 2
                INC  DPTR
                MOVX @R0, A
                MOVX A, @DPTR         ; Unroll loop 3
                INC  DPTR
                MOVX @R0, A
                MOVX A, @DPTR         ; Unroll loop 4
                INC  DPTR
                MOVX @R0, A
                MOVX A, @DPTR         ; Unroll loop 5
                INC  DPTR
                MOVX @R0, A
                MOVX A, @DPTR         ; Unroll loop 6
                INC  DPTR
                MOVX @R0, A
                MOVX A, @DPTR         ; Unroll loop 7
                INC  DPTR
                MOVX @R0, A
                MOVX A, @DPTR         ; Unroll loop 8
                INC  DPTR
                MOVX @R0, A
                MOVX A, @DPTR         ; Unroll loop 9
                INC  DPTR
                MOVX @R0, A
                MOVX A, @DPTR         ; Unroll loop 10
                INC  DPTR
                MOVX @R0, A
```

```
                    MOVX A, @DPTR         ; Unroll loop 11
                    INC  DPTR
                    MOVX @R0, A
                    MOVX A, @DPTR         ; Unroll loop 12
                    INC  DPTR
                    MOVX @R0, A
                    MOVX A, @DPTR         ; Unroll loop 13
                    INC  DPTR
                    MOVX @R0, A
                    MOVX A, @DPTR         ; Unroll loop 14
                    INC  DPTR
                    MOVX @R0, A
                    MOVX A, @DPTR         ; Unroll loop 15
                    INC  DPTR
                    MOVX @R0, A
                    MOVX A, @DPTR         ; Unroll loop 16
                    INC  DPTR
                    MOVX @R0, A
                    DJNZ R7, EP2InLoop
                    MOV  R0, #LOW(EP2InByteCount)
                    DJNZ PacketCounter, ValidateFullBuffer
                    MOV  A, LastPacketValidBytes   ; Get actual Byte Length
                    MOVX @R0, A           ; Validate the short buffer
                    CLR  Timer0_Enable    ; Stop the timer
                    CLR  B.1              ; We're done with Task1
                    RET
ValidateFullBuffer:
                    MOV  A, #64           ; Default is a full packet
                    MOVX @R0, A           ; Validate the buffer
                    JMP  WaitForBuffer    ; To prepare more data
```

Figure 11-4. Servicing the bulk endpoints in MAIN

Step 4: The BlockIO Device Driver

Windows does not include a simple USB device, bulk transfer driver so I wrote one for this example. If you do not wish to learn about writing a Windows device driver you may skip this step.

I capture the essence of writing a WDM device driver in this section. Note however, that the “devil is in the details” and I recommend that you buy Walter Oney’s book *Programming the Microsoft Windows Driver Model* if you plan to edit this, or create your own device driver. Walter takes over 600 pages to cover what I will highlight in this section.

An essential tool you will also need is the Windows 2000 Device Driver Kit. This is a free download from www.microsoft.com/hwdev. It is over 40MB and contains many interesting examples and a complete set of Include and Library files that are required to build this example.

A USB-centric WDM device driver is one of the easier device drivers to write since much of the code is provided by the operating system (or Walter Oney). Figure 11-5 shows the WDM architecture, which I shall explain, from the top down.

Figure 11-5. Windows 2000 I/O Architecture

Note that the code that we must write is depicted in ovals while the code supplied with the operating system is in boxes. To access a USB I/O device an application program makes Win32 API subroutine calls such as CreateFile, ReadFile etc. The Win32 subsystem uses a system service interface to pass this request to the I/O manager. This interface moves the program from user mode to kernel mode and there is therefore much parameter checking, data copy and other schemes to protect the kernel. The exact details of this interface need not be known since we have one Microsoft-supplied module talking to another Microsoft-supplied module. All Kernel software is object oriented which, for us, means that it uses pre-defined data structures that are manipulated with pre-defined actions. A key object (or data structure) for this discussion is the I/O Request Packet, or IRP,

that is created by the I/O manager as a result of the Win32 system call. This IRP is passed around the kernel as a message and various modules process it. The IRP is eventually returned to the I/O manager which then provides a response to the Win32 subsystem and therefore to the user application.

The Device Drivers block handles the IRP. In the general case this will involve accessing the PC host hardware via the Hardware Abstraction Layer. The complexities of dealing with re-assignable interrupt levels and direct memory access (DMA) are excessive but, as we shall see, the USB programmer does not have to deal with this. Figure 11-6 shows more detail inside the Device Drivers block.

Figure 11-6. WDM's layered device drivers

The IRP is passed to the top of the device driver stack – this could be filter driver as shown in the general model on the left or directly to the Function Driver as shown in the example on the right. An upper filter can inspect and act on any IRP before the function driver sees it and a lower filter can inspect and act on any IRP after the function driver has processed it. The function driver is what most people call the device driver and it may be a monolithic module or be constructed of a class driver plus a miniport driver. In our case, the function driver will

create USB Request Blocks, or URBs, that it will embed inside an IRP and will forward to the next lower device driver.

IRP's eventually reach the bus driver which, in our case, is the USB host controller. Microsoft supplies this USB bus driver (USBD.SYS) and does not allow programmatic access to the USB host controller. Many PC Host applications are making multiple requests to the I/O manager that directs all of the USB traffic via USBD.SYS. There is a lot of buffer management and scheduling to correctly manage the shared resource called USB. The good news is that we do not have to deal with interrupt levels or DMA and this will make our device drivers much easier to write and debug.

The device driver stack of Figure 11-6 is built from the bottom upwards and several key objects are created. As the PC host is powering up it runs several enumeration programs to discover what hardware is currently installed in your PC host. If the PCI enumerator discovers a USB host controller it will load up USBD.SYS which will create a Physical Device Object (PDO) to represent this unique hardware. If multiple USB host controllers are discovered then each will be represented by a unique PDO. Each USB is now enumerated and attached devices are identified (see Chapter 3 for this detail). Each unique device will cause a device driver to be loaded that creates a Functional Device Object (FDO). The registry is queried to discover if lower or upper filters should also be added to the stack. If two identical USB I/O devices are discovered then the device driver will handle this case (described later). The structure and location of each layer of this driver stack is saved by the I/O manager which will supply this data inside an IRP object for each message it dispatches.

I apologize if this is getting a little deep. You could stop here and be thankful that many class drivers are supplied with Windows, or, you can take a deep breath, and jump inside a driver.

It is best to think of a device driver as a set of subroutines that you must supply. Figure 11-7 shows an overview of the subroutines that are required and are optional for our USB I/O device.

Figure 11-7. A WDM driver is a set of subroutines

Once our device driver is loaded into memory the operating system calls DriverEntry – it is our responsibility to complete the DriverObject passed to us with the information about the system functions that we will supply. The DriverObject's MajorFunction table has default values for all of the subroutine entry points that are over-written by DriverEntry. The second parameter passed to DriverEntry is a pathname to a registry entry that matches this driver – if we need to access the registry then we should save this pathname. The DriverEntry routine for our BlockIO example is shown in Figure 11-8.

```
extern "C" NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
        KdPrint((DRIVERNAME "This is BETA software (V0.9) \n"));
// See if we're running under Win98 or Win2K
        win98 = IsWin98();
        KdPrint((DRIVERNAME "Running under Windows(R) "));
        if (win98) KdPrint(("98\n")); else KdPrint(("2000\n"));
// Initialize function pointers
        DriverObject->DriverUnload = DriverUnload;
        DriverObject->DriverExtension->AddDevice = AddDevice;
        DriverObject->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;
        DriverObject->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;
        DriverObject->MajorFunction[IRP_MJ_READ] = DispatchReadWrite;
        DriverObject->MajorFunction[IRP_MJ_WRITE] = DispatchReadWrite;
        DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchControl;
        DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
        DriverObject->MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL] = DispatchInternalControl;
        DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp;
        return STATUS_SUCCESS;
        }
```

Figure 11-8. The BlockIO DriverEntry routine

KdPrint is a debug routine that prints messages to a kernel debug console. I use debugview which is a free download from www.sysinternals.com.

The AddDevice routine is called by the PnP manager for each instance of identical hardware it finds. If you have three USBSIMM boards connected, for example, and their INF files all point to the same driver (this one) then AddDevice will be called three times. It is our responsibility to create an FDO for this added device and to register its existence so that application programs can find it. I prefer to use a GUID to identify my I/O devices; this unique identifier is know by the I/O device and is also know by the applications program and they use it as a rendezvous between user and kernel mode software. The AddDevice routine for our BlockIO example is shown in Figure 11-9.

```
NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT pdo) {
        PAGED_CODE();
        KdPrint((DRIVERNAME "Adding Device\n"));
        NTSTATUS status;
// Create a functional device object to represent the hardware we're managing.
        PDEVICE_OBJECT fdo;
        ULONG dxsize = (sizeof(DEVICE_EXTENSION) + 7) & ~7;
        ULONG xsize = dxsize + GetSizeofGenericExtension();
        status = IoCreateDevice(DriverObject, xsize, NULL, FILE_DEVICE_UNKNOWN,
                FILE_DEVICE_SECURE_OPEN, FALSE, &fdo);
        PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
        pdx->DeviceObject = fdo;
        pdx->Pdo = pdo;
// Declare the buffering method we'll use for read/write requests
        fdo->Flags |= DO_DIRECT_IO;
// Link our device object into the stack leading to the PDO
        pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(fdo, pdo);
// Set power management flags in the device object
        fdo->Flags |= DO_POWER_PAGABLE;
// Initialize to use the GENERIC.SYS library
        pdx->pgx = (PGENERIC_EXTENSION) ((PUCHAR) pdx + dxsize);
        GENERIC_INIT_STRUCT gis = {sizeof(GENERIC_INIT_STRUCT)};
        gis.DeviceObject = fdo;
        gis.Pdo = pdo;
        gis.Ldo = pdx->LowerDeviceObject;
        gis.RemoveLock = &pdx->RemoveLock;
        gis.StartDevice = StartDevice;
        gis.StopDevice = StopDevice;
        gis.RemoveDevice = RemoveDevice;
        RtlInitUnicodeString(&gis.DebugName, LDRIVERNAME);
        gis.Flags = GENERIC_SURPRISE_REMOVAL_OK;
        status = InitializeGenericExtension(pdx->pgx, &gis);
// Register the interface with a GUID
        status = GenericRegisterInterface(pdx->pgx, &BlockIO_GUID);
// Clear the "initializing" flag so that we can get IRPs
        fdo->Flags &= ~DO_DEVICE_INITIALIZING;
        return status;
        }
```

Figure 11-9. The BlockIO AddDevice routine

There is no error checking in Figure 11-9, I did this to reduce the size of the figure – the version on the CDROM has extensive error checking and should be used for development. Many of the tasks that must be implemented in a device driver are very specialized but are always the same code for every device. To reduce the amount of difficult code that we have to write, Walter Oney has supplied a GENERIC.SYS driver that implements the "tricky pieces" such as power management and the instrumentation interfaces. He supplies the source code of the generic.sys helper functions for those readers who **really** want to know what is going on (I did peek at the code, but then decided to trust the expert). Our AddDevice routine creates links into the generic.sys library.

At this stage of the events our I/O device driver has been loaded and is ready for action. If we were to remove the I/O device at this time then the driver would be unloaded and the FDO deleted. If we replug our USB I/O device then the PnP manager enumerates it again, the driver is reloaded and started via its DriverEntry routine and the PnP manager will call AddDevice again. Nothing else happens until an application program tries to use this device driver; so lets move our attention to there for a moment.

An application program will use the same search algorithm that I presented in Chapter 6. This program makes system calls to the PnP manager and requests a list of attached devices; we then search through this list for our particular I/O device. The only difference between this BlockIO example and the HID example of Chapter 6 is that we use a different GUID. The searching will identify a system name which the application program will use in a CreateFile Win32 API call- this will be delivered to our device driver as a call to StartDevice.

The StartDevice routine for our BlockIO example is shown in Figure 11-10. Again there is no error checking in the figure to save space, the real source code on the CDROM contains extensive error checking and the support routines. The routine interrogates the I/O device to ensure that it has the correct endpoints and then opens a pipe to each of these endpoints. The I/O manager will create a HANDLE that is will pass back to the application program so that subsequent reads and writes will manipulate the correct pipes.

```
NTSTATUS StartDevice(PDEVICE_OBJECT fdo, PCM_PARTIAL_RESOURCE_LIST raw,
PCM_PARTIAL_RESOURCE_LIST translated) {
	PAGED_CODE();
	KdPrint((DRIVERNAME "Starting Device\n"));
	NTSTATUS status;
	PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
	URB urb;

// Read our device descriptor. Check that we have the correct interface and get a handle for each pipe
	UsbBuildGetDescriptorRequest(&urb, sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
		USB_DEVICE_DESCRIPTOR_TYPE, 0, 0, &pdx->dd, NULL, sizeof(pdx->dd), NULL);
```

```
        status = SendAwaitUrb(fdo, &urb);
        MSGUSBSTRING(fdo, DRIVERNAME " - Configuring device from '%ws'\n", pdx->dd.iManufacturer);
// Get Product Name.  This is used to identify multiple BlockIO devices
        UNICODE_STRING ProductName;
        status = GetStringDescriptor(fdo, pdx->dd.iProduct, &ProductName);
        pdx->ProductName.Buffer = ProductName.Buffer;
        pdx->ProductName.MaximumLength = ProductName.MaximumLength;
        RtlCopyUnicodeString(&pdx->ProductName, &ProductName);
        RtlFreeUnicodeString(&ProductName);
        KdPrint((DRIVERNAME "Product name is '%ws'\n", &pdx->ProductName.Buffer));

// Read the descriptor of the first configuration in two steps. First read the configuration descriptor header.
// The second step reads the configuration descriptor plus all embedded interface and endpoint descriptors.
        USB_CONFIGURATION_DESCRIPTOR tcd;
        UsbBuildGetDescriptorRequest(&urb, sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
                USB_CONFIGURATION_DESCRIPTOR_TYPE, 0, 0, &tcd, NULL, sizeof(tcd), NULL);
        status = SendAwaitUrb(fdo, &urb);
        ULONG size = tcd.wTotalLength;
        PUSB_CONFIGURATION_DESCRIPTOR pcd = (PUSB_CONFIGURATION_DESCRIPTOR)
                ExAllocatePool(NonPagedPool, size);
        UsbBuildGetDescriptorRequest(&urb, sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
                USB_CONFIGURATION_DESCRIPTOR_TYPE, 0, 0, pcd, NULL, size, NULL);
        status = SendAwaitUrb(fdo, &urb);
        MSGUSBSTRING(fdo, DRIVERNAME "Selecting configuration named '%ws'\n", pcd->iConfiguration);

// Locate the descriptor for the one and only interface we expect to find
        PUSB_INTERFACE_DESCRIPTOR pid = USBD_ParseConfigurationDescriptorEx(pcd, pcd, -1, -1, -1, -1, -1);
        MSGUSBSTRING(fdo, DRIVERNAME "Selecting interface named '%ws'\n", pid->iInterface);

// Create a URB to use in selecting a configuration.
        USBD_INTERFACE_LIST_ENTRY interfaces[2] = {
                {pid, NULL},
                {NULL, NULL},                           // fence to terminate the array
                };

        PURB selurb = USBD_CreateConfigurationRequestEx(pcd, interfaces);
        PUSBD_INTERFACE_INFORMATION pii = interfaces[0].Interface;
// Initialize the maximum transfer size for each of the endpoints
        pii->Pipes[0].MaximumTransferSize = TRANSFER_SIZE;
        pii->Pipes[1].MaximumTransferSize = TRANSFER_SIZE;
        pdx->maxtransfer = TRANSFER_SIZE;          // save for use in handling reads & writes
// Submit the set-configuration request
        status = SendAwaitUrb(fdo, selurb);
// Save the configuration and pipe handles
        pdx->hconfig = selurb->UrbSelectConfiguration.ConfigurationHandle;
        pdx->hinpipe = pii->Pipes[0].PipeHandle;
        pdx->houtpipe = pii->Pipes[1].PipeHandle;
// Transfer ownership of the configuration descriptor to the device extension
        pdx->pcd = pcd;
        pcd = NULL;
        return STATUS_SUCCESS;
        }
```

Figure 11-10. The BlockIO StartDevice routine

Another interesting facet of this device driver is the handling of ReadFile and WriteFile that both result in the DispatchReadWrite routine being called. The DispatchReadWrite routine breaks the user buffer into 4KB blocks and submits these, one at a time, to USBD.SYS. While this reduces the maximum data rate for our I/O device slightly it does improve the system performance since other I/O requests can be interleaved between these 4KB blocks. This routine is paired with the OnReadWriteComplete routine that checks the success of the transaction and resubmits the IRP with updated buffer pointer values until the entire transfer has been completed. These routines are shown in Figure 11-11, again with minimal error checking.

```
NTSTATUS DispatchReadWrite(PDEVICE_OBJECT fdo, PIRP Irp) {
        PAGED_CODE();
        PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
        NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
        PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
        BOOLEAN read = stack->MajorFunction == IRP_MJ_READ;

        USBD_PIPE_HANDLE hpipe = read ? pdx->hinpipe : pdx->houtpipe;
        LONG haderr;
        if (read) haderr = InterlockedExchange(&pdx->inerror, 0);
        else haderr = InterlockedExchange(&pdx->outerror, 0);
        if (haderr && !NT_SUCCESS(ResetPipe(fdo, hpipe))) ResetDevice(fdo);

// Our strategy will be to do the transfer in stages up to MAXTRANSFER bytes
// long using a single URB that we resubmit during the completion routine.
// Note that chained URBs via UrbLink are not supported in either Win98 or Win2K.
        PRWCONTEXT ctx = (PRWCONTEXT) ExAllocatePool(NonPagedPool, sizeof(RWCONTEXT));
        RtlZeroMemory(ctx, sizeof(RWCONTEXT));

        ULONG length = Irp->MdlAddress ?  MmGetMdlByteCount(Irp->MdlAddress) : 0;
        KdPrint((DRIVERNAME "In DispatchReadWrite with Length = %8.8x (%d)\n", length, length));
        if (!length) {                                          // zero-length read
                IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
                return CompleteRequest(Irp, STATUS_SUCCESS, 0);
                }
        ULONG_PTR va = (ULONG_PTR) MmGetMdlVirtualAddress(Irp->MdlAddress);
        ULONG urbflags =  USBD_SHORT_TRANSFER_OK | (read ? USBD_TRANSFER_DIRECTION_IN :
                USBD_TRANSFER_DIRECTION_OUT);

// Calculate the length of the segment we'll transfer in the first stage.
        ULONG seglen = length;
        if (seglen > pdx->maxtransfer) seglen = pdx->maxtransfer;

// Allocate an MDL for the first segment of the transfer.
        PMDL mdl = IoAllocateMdl((PVOID) va, pdx->maxtransfer, FALSE, FALSE, NULL);

// Initialize the (partial) MDL to describe the first segment's subset of the user buffer.
        IoBuildPartialMdl(Irp->MdlAddress, mdl, (PVOID) va, seglen);
        UsbBuildInterruptOrBulkTransferRequest(ctx, sizeof(_URB_BULK_OR_INTERRUPT_TRANSFER),
                hpipe, NULL, mdl, seglen, urbflags, NULL);

// Set context structure parameters to pick up where we will leave off
        ctx->va = va + seglen; ctx->length = length - seglen; ctx->mdl = mdl; ctx->numxfer = 0;
```

```
// Use the original Read or Write IRP as a container for the URB
        stack = IoGetNextIrpStackLocation(Irp);
        stack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
        stack->Parameters.Others.Argument1 = (PVOID) (PURB) ctx;
        stack->Parameters.DeviceIoControl.IoControlCode = IOCTL_INTERNAL_USB_SUBMIT_URB;

        IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE) OnReadWriteComplete,
                (PVOID) ctx, TRUE, TRUE, TRUE);
        IoMarkIrpPending(Irp);
        status = IoCallDriver(pdx->LowerDeviceObject, Irp);
        if (!NT_SUCCESS(status) && !NT_SUCCESS(ResetPipe(fdo, hpipe))) ResetDevice(fdo);

        return STATUS_PENDING;
        }

NTSTATUS OnReadWriteComplete(PDEVICE_OBJECT fdo, PIRP Irp, PRWCONTEXT ctx) {
        PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
        BOOLEAN read = (ctx->UrbBulkOrInterruptTransfer.TransferFlags &
                USBD_TRANSFER_DIRECTION_IN) != 0;
        ctx->numxfer += ctx->UrbBulkOrInterruptTransfer.TransferBufferLength;
        USBD_STATUS urbstatus = URB_STATUS(ctx);

// If this stage completed without error, resubmit the URB to perform the next stage
        NTSTATUS status = Irp->IoStatus.Status;
        if (NT_SUCCESS(status) && ctx->length) {
// Calculate length of next stage of the transfer
                ULONG seglen = ctx->length;
                if (seglen > pdx->maxtransfer) seglen = pdx->maxtransfer;

// We're now in arbitrary thread context, so the virtual address of the user buffer is currently meaningless.
// IoBuildPartialMdl copies physical page numbers from the original IRP's probed-and-locked MDL, however,
// so our process context doesn't matter.
                PMDL mdl = ctx->mdl;
                MmPrepareMdlForReuse(mdl);
                IoBuildPartialMdl(Irp->MdlAddress, mdl, (PVOID) ctx->va, seglen);
// Reinitialize the URB
                ctx->UrbBulkOrInterruptTransfer.TransferBufferLength = seglen;
// The "next" stack location is the one belonging to the driver underneath us.
// It's been mostly set to zero and must be reinitialized.
                PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);
                stack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
                stack->Parameters.Others.Argument1 = (PVOID) (PURB) ctx;
                stack->Parameters.DeviceIoControl.IoControlCode = IOCTL_INTERNAL_USB_SUBMIT_URB;
                IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE) OnReadWriteComplete,
                        (PVOID) ctx, TRUE, TRUE, TRUE);
                ctx->va += seglen; ctx->length -= seglen;
// Pass the recycled IRP down and ignore the return code from IoCallDriver.
                IoCallDriver(pdx->LowerDeviceObject, Irp);
                return STATUS_MORE_PROCESSING_REQUIRED; // halt completion process in its tracks!
                }
// The request is complete now
        Irp->IoStatus.Information = ctx->numxfer;
        IoFreeMdl(ctx->mdl);
        ExFreePool(ctx);
        IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
        return status;
        }
```

Figure 11-11. The BlockIO DispatchReadWrite and OnReadWriteComplete routines

I wanted to “open” my BlockIO device using its name as I did in the HID example. I had to create an equivalent of GetProductName for my BlockIO device – I did this with a customer control code that is parsed by the DispatchControl routine (Figure 11-12).

```
NTSTATUS DispatchControl(PDEVICE_OBJECT fdo, PIRP Irp) {
        PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
        PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
        NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
        if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
// Check for my "WhoAreYou" IOCTL
        ULONG IoControlCode = stack->Parameters.DeviceIoControl.IoControlCode;
        KdPrint((DRIVERNAME "IOCTL %4.4x detected\n", IoControlCode));
        if (IoControlCode == IOCTL_GET_PRODUCT_NAME) {
            ULONG BufferLength = stack->Parameters.DeviceIoControl.OutputBufferLength;
            ULONG ActualLength = 0; // Default
            if (BufferLength < pdx->ProductName.Length) status = STATUS_INVALID_BUFFER_SIZE;
            else {
                ActualLength = pdx->ProductName.Length;
                memcpy(Irp->AssociatedIrp.SystemBuffer, pdx->ProductName.Buffer, ActualLength);
                status = STATUS_SUCCESS;
                }
            IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
            return CompleteRequest(Irp, status, ActualLength);
            }
        else { // Not mine, pass this down the stack
            IoSkipCurrentIrpStackLocation(Irp);
            status = IoCallDriver(pdx->LowerDeviceObject, Irp);
            IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
            return status;
            }
        }
```

Figure 11-12. Adding a custom IOControl using a DispatchControl routines

Other routines, including DispatchClose and a variety of pipe handling functions are also required. The full driver source in Visual C++ project file format is on the CDROM. The executable file is called blockio.sys and is copied into the Windows drivers directory.

Step 5. The Application Program

I chose a simple, prompted, Visual C console program to test this driver.

```
bool OpenUSBinterfaces() {
		struct _GUID GUID[2];
		SP_INTERFACE_DEVICE_DATA DeviceInterfaceData;
		struct {DWORD cbSize; char DevicePath[256];} FunctionClassDeviceData;
		int Device, Interface, i;
		HANDLE PnPHandle;
		char buffer[256];
		unsigned long BytesReturned;
		bool Success, Openned;
		const char *DeviceName = "ECHO1";
// Initialize the GUID array
		HidD_GetHidGuid(&GUID[0]);
		memcpy(&GUID[1], &BlockIO_GUID, sizeof(BlockIO_GUID));
// Our device has two interfaces, 0 = HID and 1 = BlockIO.  Open them both
		BytesReturned = IOCTL_GET_PRODUCT_NAME;
		for (Interface = 0; Interface<2; Interface++) {
// Get a handle for the Plug and Play node and request currently active devices
			PnPHandle = SetupDiGetClassDevs(&GUID[Interface], NULL, NULL,
				DIGCF_PRESENT | DIGCF_INTERFACEDEVICE);
			if (int(PnPHandle) == -1) {printf("Could not attach to PnP node"); return false; }
// Lets look for a maximum of 20 Devices
			Handle[Interface] = INVALID_HANDLE_VALUE;
			for (Device = 0; (Device < 20); Device++) {
// Initialize our data
				DeviceInterfaceData.cbSize = sizeof(DeviceInterfaceData);
// Is there a device at this table entry
				Success = SetupDiEnumDeviceInterfaces(PnPHandle, NULL, &GUID[Interface],
					Device, &DeviceInterfaceData)
				if (Success) {
// There is a device here, get it's name
					FunctionClassDeviceData.cbSize = 5;
					Success = SetupDiGetDeviceInterfaceDetail(PnPHandle, &DeviceInterfaceData,
						(PSP_INTERFACE_DEVICE_DETAIL_DATA)&FunctionClassDeviceData,
						256, &BytesReturned, NULL);
					if (!Success) {printf("Could not find the system name for this Device\n"); return false; }
// Can now open this Device
					Handle[Interface] = CreateFile(FunctionClassDeviceData.DevicePath, _
						0xC0000000, 3, NULL, 3, 0, NULL);
// Is it OUR Device?
					if (Interface == 0) HidD_GetProductString(Handle[Interface], buffer, sizeof(buffer));
					else DeviceIoControl(Handle[Interface], IOCTL_GET_PRODUCT_NAME,
						NULL, 0, buffer, sizeof(buffer), &BytesReturned, NULL);
// Compare incoming string with UNICODE string
					Openned = true; i = 0;
					while (DeviceName[i] != 0) {if (buffer[2*i] != DeviceName[i]) {Openned = false;} i++;}
					printf("%s Interface %sfound\n", Interface ? "BlockIO" : "HID", Openned ? "" : "not ");
					}
				} // if (SetupDiEnumDeviceInterfaces . .
			} // for (Device = 0; (Device < 20); Device++)
		CloseHandle(PnPHandle);
		return true;
}
```

```
int main(int argc, char* argv[]) {
		char InputCharacter;
		int Choice, BufferSize, LEDvalue, i;
		unsigned long BytesWritten, BytesRead;
		UCHAR DataBuffer[0x3FFF]; // Maximum size for 'ECHO1' firmware
		UCHAR ButtonsCommand[4] = {0,1,0,0};
		UCHAR LightsCommand[4] = {0,2,0,0};
		UCHAR SendCommand[4] = {0,3,0,0};
		UCHAR ReadCommand[4] = {0,4,0,0};
		UCHAR ReplyBuffer[2];

		printf("\nTest program for 'ECHO1' HIDwithBlockIO Example\n\n");
		OpenUSBinterfaces());
		HID = Handle[0]; BlockIO = Handle[1]; // Make program easier to read
		if ((HID == INVALID_HANDLE_VALUE) || (BlockIO == INVALID_HANDLE_VALUE)) {
				CloseHandle(HID);
				CloseHandle(BlockIO);
				printf("\nNeed both interfaces open for the 'ECHO1' device\n");
				printf("\nERROR EXIT\n");
				cin >> InputCharacter;
				return -1;
				}
// Put something recognizable into the data buffer
		for (i=0; i<sizeof(DataBuffer); i++) DataBuffer[i] = (UCHAR) i;
		BufferSize = 0x3FFF;
		do {		printf("\nThe following commands are available:\n");
				printf("B = Read the device 'Buttons'\n");
				printf("L = Set the device 'Lights'\n");
				printf("C = Change the data buffer size (%d)\n", BufferSize);
				printf("S = Send the data buffer\n");
				printf("R = Receive the data buffer\n");
				printf("E = Exit this program\n");
				printf("Please enter your choice - ");
				cin >> InputCharacter;
				Choice = InputCharacter;
				switch (tolower(Choice)) {
					case 'b':
						if (!WriteFile(HID, ButtonsCommand, sizeof(ButtonsCommand), &BytesWritten, 0))
							printf("Error writing Command to I/O device\n");
						if (!ReadFile(HID, ReplyBuffer, sizeof(ReplyBuffer), &BytesRead, NULL))
							printf("Error reading from HID device\n");
						printf("\nButtons value = ");
						for (i=0; i<8; i++) printf("%s", ((ReplyBuffer[1] >> i) & 1) ? "0" : "1");
						printf("\n");
						break;
					case 'l':
						printf("\nValue to be written to Lights: ");
						cin >> LEDvalue;
						LightsCommand[2] = LEDvalue;
						if (!WriteFile(HID, LightsCommand, sizeof(LightsCommand), &BytesWritten, NULL))
							printf("Error writing Command to I/O device\n");
						break;
					case 'c':
						printf("\nEnter new value for buffer size: ");
						cin >> BufferSize;
						if (BufferSize > 0x3FFF) {
							BufferSize = 0x3FFF;
							printf("Maximum buffer size for 'ECHO1' device is %d \n", 0x3FFF);
							}
						break;
```

```
            case 's':
                printf("\nSending the data buffer\n");
                if (!WriteFile(HID, SendCommand, sizeof(SendCommand), &BytesWritten, NULL))
                    printf("Error writing Command to I/O device\n");
                if (WriteFile(BlockIO, DataBuffer, BufferSize, &BytesWritten, NULL))
                    printf("Write transferred %d bytes okay\n", BytesWritten);
                else printf("Error %d trying to write BlockIO data\n", GetLastError());
                break;
            case 'r':
                printf("\nReceiving the data buffer\n");
                ReadCommand[2] = BufferSize & 0xFF; ReadCommand[3] = BufferSize >> 8;
                if (!WriteFile(HID, ReadCommand, sizeof(ReadCommand), &BytesWritten, NULL))
                    printf("Error writing Command to I/O device\n");
                if (ReadFile(BlockIO, DataBuffer, BufferSize, &BytesRead, NULL))
                    printf("Read transferred %d bytes correctly\n", BytesRead);
                else printf("Error %d trying to read BlockIO data\n", GetLastError());
                break;
            case 'e':
                printf("\n\n");
                CloseHandle(HID);
                CloseHandle(BlockIO);
                return 0;
            default: printf("\nInvalid Selection\n"); break;
            }
        } while (1);
} // Main
```

Figure 11-13. MAIN routine for BlockIO Test program

The application code searches for the device twice – once to identify and open the HID interface and once to identify and open the BlockIO interface. Note in Figure 11-13 that reading and writing the BlockIO interface is identical to reading writing the HID interface except, of course, the SIZE of the data reads/writes.

Step 6: Installing our BlockIO Device

This BlockIO example has two interfaces therefore it requires two INF files to specify two separate device drivers. The HID interface will be matched by the default HIDDEV.INF file provided with Windows. The BlockIO interface requires a specific INF file as shown in Figure 11-14.

```
[Version]
Signature=$CHICAGO$
Class=UNKNOWN
Provider=%MFGNAME%

[Manufacturer]
%MFGNAME%=DeviceList

[DestinationDirs]
DefaultDestDir=10,System32\Drivers

[SourceDisksFiles]
blockio.sys=1
generic.sys=1

[SourceDisksNames]
1=%INSTDISK%,,,

[DeviceList]
%DESCRIPTION%=DriverInstall,USB\VID_4242&PID_4220&MI_01

;  Windows 2000 Sections
[DriverInstall.NT]
CopyFiles=DriverCopyFiles

[DriverCopyFiles]
blockio.sys,,,2
generic.sys,,,2

[DriverInstall.NT.Services]
AddService=BlockIO,2,DriverService

[DriverService]
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%10%\system32\drivers\blockio.sys

[DriverInstall.nt.hw]
AddReg=DriverHwAddReg

[DriverHwAddReg]
HKR,,FriendlyName,,%FRIENDLYNAME%

;  Windows 98 Sections
[DriverInstall]
AddReg=DriverAddReg
CopyFiles=DriverCopyFiles

[DriverAddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,blockio.sys

[DriverInstall.hw]
AddReg=DriverHwAddReg

;  String Definitions
 [Strings]
MFGNAME="USB Design By Example"
INSTDISK="USB Design By Example, Examples Disk"
DESCRIPTION="HIDwithBlockIO Example Device"
FRIENDLYNAME="Simple Block IO Driver"
```

Figure 11-14 Block IO INF file

Since windows does not have a device class called BlockIO I created one and also a GUID using GUIDGEN.exe from the Windows 2000 DDK. The unique GUID allows the application program in Step 5 to search for and open the BlockIO.sys device driver created in Step 4.

Install the microcontroller firmware created in Step 3 onto the EZ-USB development board using the dScope debugger. Once this firmware starts executing it will re-numerate and the Windows hardware installation wizard will first install the HID drivers and will then install the BULKIO device driver.

The application program created in Step 5 can now be started – it will search for and find the BlockIO hardware device. Buffers of data can be transferred to and from the I/O device, and the Buttons and Lights still work!

BLOCKIO EXAMPLE SUMMARY

We have successfully moved blocks of data, as fast as possible, between a PC host and an I/O device. The I/O device firmware used for the "Buttons and Lights" example was extended to add bulk in and bulk out endpoints. A BULKIO.sys device driver was created to manage the data transfer between an application program and the USB device driver. The full source of this device driver is provided on the CDROM and may be used as is or modified to suit your particular requirements. An application program was written to test the block IO data transfers. The source of the application program is provided on the CDROM in Visual C++ project files and the matching firmware is saved as a Keil assembler project.

WELCOME TO THE KERNEL WORLD

For those of you who have enjoyed our brief journey into the world of kernel programming I invite you to stay and look around a little. In this section I explore other device drivers including some filters. Or you can skip this section and move on to “Using a Windows-supplied device driver”.

I was a little wary including this material in the book since the realm of device drivers is not very forgiving. Program errors can hang your system. Bad variable declarations can crash your operating system. Treat this layer with the same respect that you would a power tool such as a circular saw – it can rip wood much faster than a hand saw, but it can take your arm off just as easily!

To reduce our risk in this world I will use template driver programs developed by Walter Oney and provided with his permission. If you want to learn the theory about filter drivers, and device drivers in general, I recommend that you purchase his book, *Programming the Windows Driver Model* (ISBN 0-7356-0518-1)

EZ-USB Autoloader

My second device driver, LoadEz.sys, is actually simpler than the BlockIO.SYS example. It is a firmware autoloader for the EZ-USB component. This driver must detect that an EzUSB-based I/O device is being attached to the PC host and then interrogate this device to discover its VID and PID values. The driver will then download firmware onto this EZ-USB board depending upon the values of VID_PID. There are many examples in this book and each has a different PID (they all have the same VID) so different firmware will be loaded into each example.

The INF file for each VID-PID combination will identify LoadEZ.sys as the device driver for their board. The firmware will be downloaded and then the board will renumerate with a different VID and PID to represent the new identity of the loaded firmware. DO NOT USE the same VID and PID since this will put your I/O device into an infinite loop of reloading and renumerating.

This LoadEZ.sys autoloader has boilerplate DeviceEntry and AddDevice routines. The work gets done in the StartDevice PnP system call as shown in Figure 11-15

```
NTSTATUS StartDevice(PDEVICE_OBJECT fdo, PCM_PARTIAL_RESOURCE_LIST raw,
		PCM_PARTIAL_RESOURCE_LIST translated) {
	PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
	KdPrint((DRIVERNAME "Starting IO device\n"));
// Open the HEX file and read it into a local buffer
	NTSTATUS Status = GetFirmware(fdo);
// Loading is done in two passes since AnchorLoad only operates on internal memory addresses
// Load external memory first using a helper "mover" program
// First ensure that the EZ-USB 8051 is RESET before downloading
	Status = EZUSB_Reset(fdo, 1);
	UCHAR MoverCode[] =	// 8051 Object code for the Mover program
		{0x90, 0x7F, 0x9E, 0x74, 0x0C0, 0x0F0, 0x90, 0x7F, 0x95, 0x0F0, 0x75, 0x92, 0,
		0x78, 0x24, 0x0E2, 0x60, 0x10, 0x0FF, 8, 0x0E2, 0xF5, 0x83, 8, 0x0E2, 0x0F5, 0x82,
		8, 8, 0x0E2, 0x0F0, 0x0A3, 0x0DF, 0x0FA, 0x80, 0x0FE, 0x0F, 0x0FF, 0x0F0, 0, 0x4D,
		0x6F, 0x76, 0x65, 0x72, 0x20, 0x49, 0x6E, 0x73, 0x74, 0x61, 0x6C, 0x6C, 0x65, 0x64};
	SetEZUSBMemory(fdo, 0, sizeof(MoverCode), MoverCode);
	EZUSB_Reset(fdo, 0);			// unRESET the EZUSB CPU so that it executes this code
	UCHAR Record[40]; char Buffer[80]; int Pass, i;
	for (Pass = 1; Pass<3; Pass++) {
		pdx->ByteOffset = 0;		// Start at the beginning of the Hex data file
		EZUSB_Reset(fdo, 1);		// Stop the EZUSB CPU
		USHORT ByteCount, LoadAddress, RecordType;
// Read the HEX records one at a time and load them onto the development board
		while (GetNextLine(fdo, Buffer)) {
			ByteCount = Record[0] = (value(Buffer[1]) << 4) + value(Buffer[2]);
			for (i = 1; i<ByteCount+4; i++) Record[i] = (value(Buffer[1+i+i]) << 4) + value(Buffer[2+i+i]);;
			LoadAddress = (Record[1] << 8) + Record[2];
			RecordType = Record[3];
			if (RecordType == 0) {
// A content record (=0) has been located
				if ((Pass == 1) && (LoadAddress > 0x1FFF)) {
					SetEZUSBMemory(fdo, 0x24, ByteCount+4, Record);
					EZUSB_Reset(fdo, 0);	// unRESET EZUSB CPU so that it will move the data
					EZUSB_Reset(fdo, 1);	// Stop the EZUSB CPU again
					}
				if ((Pass == 2) && (LoadAddress + ByteCount) < 0x1B3F) {
					SetEZUSBMemory(fdo, LoadAddress, ByteCount, &Record[4]);
					}
				}
			}
		}
// All done, allow the EZ-USB CPU to 'renumerate'
	KdPrint((DRIVERNAME "Firmware downloaded!\n"));
	EZUSB_Reset(fdo, 0);
// Interesting philosophical point - the original device that caused this driver to run no longer
// exists, since the driver has downloaded it with a new identity.
// It would be preferrable to return UNSUCCESSFUL so the OS will mark us STOPPED.
// This also prevents "Surprize Removal" messages on Win2000
// However, the USDB.SYS driver will Suspend our I/O device if we do.  So return SUCCESS
	return STATUS_SUCCESS;
	}
```

Figure 11-15 The work is done in StartDevice routine

The first thing the GetFirmware routine (Figure 11-16) needs to do is to identify this particular EZ-USB board. The GetIdentity routine (Figure 11-17) needs to create a USB Request Block and submit it to the kernel for processing. One wrinkle with the Win98 solution was due to the order of subsystem initialization following a power up. The PnP manager calls our AddDevice routine BEFORE the Windows file system has been initialized – which means that a CreateFile to open the firmware hex file will fail. Walter Oney came to the rescue again and showed me how to access the DOS file drivers underneath Windows!

```
NTSTATUS GetFirmware(PDEVICE_OBJECT fdo) {
        PAGED_CODE();
// Open the Firmware file and load it into a local buffer
// First create the name of the file that we need to open
        PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
        NTSTATUS Status;
        Status = GetIdentity(fdo);

// Create a filename from the current VID and PID
        WCHAR Filename[] = L"\\SystemRoot\\system32\\drivers\\vvvvpppp.hex";
        Filename[29] = HexCharacter[(pdx->VID >> 12) & 0x0F];
        Filename[30] = HexCharacter[(pdx->VID >> 8) & 0x0F];
        Filename[31] = HexCharacter[(pdx->VID >> 4) & 0x0F];
        Filename[32] = HexCharacter[pdx->VID & 0x0F];
        Filename[33] = HexCharacter[(pdx->PID >> 12) & 0x0F];
        Filename[34] = HexCharacter[(pdx->PID >> 8) & 0x0F];
        Filename[35] = HexCharacter[(pdx->PID >> 4) & 0x0F];
        Filename[36] = HexCharacter[pdx->PID & 0x0F];
// Need to use Walter Oney's portable file subsystem here since this driver may be running before
// the file system is running
        HANDLE FileHandle;
        Status = OpenFile(Filename, TRUE, &FileHandle);
        KdPrint((DRIVERNAME "OpenFile returned with status = %8.8x\n", Status));
        if (!NT_SUCCESS(Status)) return Status;

        pdx->HexFileLength = (ULONG) GetFileSize(FileHandle);
        pdx->HexFileBuffer = (PCHAR) ExAllocatePool(NonPagedPool, pdx->HexFileLength);
        if (!pdx->HexFileBuffer) return STATUS_NO_MEMORY;
        ULONG BytesReturned;
        Status = ReadFile(FileHandle, pdx->HexFileBuffer, pdx->HexFileLength, &BytesReturned);
        if (!NT_SUCCESS(Status)) KdPrint((DRIVERNAME "ReadFile failed with status = %8.8x\n", Status));
        else KdPrint((DRIVERNAME "ReadFile returned %d bytes\n", BytesReturned));
        CloseFile(FileHandle);
        return Status;
        }
```

Figure 11-16 GetFirmware routine

```
NTSTATUS GetIdentity(PDEVICE_OBJECT fdo) {
        PAGED_CODE();
// Need to discover the VID and PID currently in use
        PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
        USB_DEVICE_DESCRIPTOR Descriptor;
        NTSTATUS Status;
        URB LocalUrb;
        PURB Urb = &LocalUrb;
        RtlZeroMemory(Urb, sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST));
        Urb->UrbHeader.Length = sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST);
        Urb->UrbHeader.Function = URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE;
        Urb->UrbControlDescriptorRequest.TransferBufferLength = sizeof(Descriptor);
        Urb->UrbControlDescriptorRequest.TransferBuffer = &Descriptor;
        Urb->UrbControlDescriptorRequest.TransferBufferMDL = NULL;
        Urb->UrbControlDescriptorRequest.DescriptorType = 1;
        Status = SendUrbAndWait(fdo, Urb);
        pdx->VID = Descriptor.idVendor;
        pdx->PID = Descriptor.idProduct;
        KdPrint((DRIVERNAME "Current VID = %4.4x and PID = %4.4x\n", pdx->VID, pdx->PID));
        return Status;
        }
```

Figure 11-17 GetIdentity routine

The hex file containing the firmware must be parsed twice due to the operation of the “Anchor Load” feature of the EZ-USB component. The Anchor Load function can only load internal memory so a helper program is needed to load external memory locations. The helper program is basically a block-move program that accepts records loaded into low internal memory and moves them to their correct location. The helper program and all data is loaded using USB Request Blocks containing the Anchor Load vendor command. Finally the EZ-USB is RESET so that it executes the loaded program and reconnects itself with new personality.

The operating system will see the initial I/O device be disconnected and will unload the LoadEZ.sys driver.

The full source code and additional instructions for the LoadEZ.sys filter program are included on the CDROM.

A Filter Driver

A filter driver sits in between two device driver layers and passes the system calls down to the lower layer and passes system responses back up to the upper layer. A NULL filter driver, that is, one that does nothing except pass system calls on and system responses back is quite a lot of software that is organized in blocks as shown in Figure 11-18.

Figure 11-18 Structure of a NULL filter driver

The main difference between a filter driver and a device driver is that a filter driver must handle ALL of the messages that the operating system could send to it (a device driver typically only implements the subset that it must handle).

A device/filter driver will be passed four types of system messages that need to be handled:

Majorfunctions – these are the run-time functions used to implement operations such as open and close.

Power Control – the operating system manages system power to a minimum wherever possible. Un-used I/O devices are instructed to power down and they may wake-up a sleeping PC if enabled.

PlugAndPlay – the operating system supports the dynamic addition and removal of I/O devices and these system calls manage this function.

Instrumentation – the operating system gathers statistics on running I/O devices for diagnostic and tuning purposes.

In this NULL filter driver all of these messages will be simply passed down to a lower device driver (Figure 11-19)

```
NTSTATUS PassDownMajorFunctions(IN PDEVICE_OBJECT fido, IN PIRP Irp) {
	PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
	UCHAR function = stack->MajorFunction;
	KdPrint((DRIVERNAME "Passing down Major function %2.2X\n", function));
	PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
	NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
	if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
	IoSkipCurrentIrpStackLocation(Irp);
	status = IoCallDriver(pdx->LowerDeviceObject, Irp);
	IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
	return status;
	}
NTSTATUS PassDownPower(IN PDEVICE_OBJECT fido, IN PIRP Irp) {
	PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
	PoStartNextPowerIrp(Irp);			// must be done while we own the IRP
	NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
	if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
	IoSkipCurrentIrpStackLocation(Irp);
	status = PoCallDriver(pdx->LowerDeviceObject, Irp);
	IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
	return status;
	}
NTSTATUS PassDownPnp(IN PDEVICE_OBJECT fido, IN PIRP Irp) {
	PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
	ULONG fcn = stack->MinorFunction;
	PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
	NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
	if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
	IoSkipCurrentIrpStackLocation(Irp);
	status = IoCallDriver(pdx->LowerDeviceObject, Irp);
	if (fcn == IRP_MN_REMOVE_DEVICE) {
		IoReleaseRemoveLockAndWait(&pdx->RemoveLock, Irp);
		RemoveDevice(fido);
		}
	else IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
	return status;
	}
NTSTATUS PassDownWmi(IN PDEVICE_OBJECT fido, IN PIRP Irp) {
	PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
	NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
	if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
	IoSkipCurrentIrpStackLocation(Irp);
	status = IoCallDriver(pdx->LowerDeviceObject, Irp);
	IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
	return status;
	}
```

Figure 11-19. A NULL filter passes all messages down

Once this example is compiled and built it creates a filter driver called NULLfilt.sys. This needs to be associated with an I/O device so that the operating system will load it. Figure 11-20 shows how to add a filter to the driver section of an INF file.

```
; NULLfilter.INF   Derived from HIDDEV.INF
[Version]
Signature="$CHICAGO$"
Class=HIDClass
Provider=%USBDBE%
LayoutFile=layout.inf

[DestinationDirs]
DefaultDestDir = 10,system32\drivers

[Manufacturer]
%USBDBE%=USBDBE

[USBDBE]
%HID.DeviceDesc% = HID_Inst,GENERIC_HID_DEVICE,USB\VID_0618&PID_0300

; Win98 Installation sections
[HID_Inst]
CopyFiles=HID_Inst.CopyFiles
AddReg=HID_Inst.AddReg

[HID_Inst.CopyFiles]
hidusb.sys
hidparse.sys
hidclass.sys
NULLfilter.sys

[HID_Inst.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,"hidusb.sys"

[HID_Inst.HW]
AddReg=HID_Inst.AddReg.HW

[HID_Inst.AddReg.HW]
HKR,,"LowerFilters",0x00010000,"NULLfilt.sys"

[strings]
USBDBE            = "USB Design By Example"
HID.DeviceDesc    = "Keypad with Debug Filter"
```

Figure 11-20. Specifying a lower level filter

When a new device is added to a running PC system a registry entry will be built using this INF file and NULLfilt.sys will be installed with the device driver. During run-time no operational changes will be visible unless you have a kernel debug console running (such as Debugview).

DISPLAY USB ENUMERATION TRANSACTIONS

The basic idea behind this example is to extend the debug display messages to give more information about the USB-specific system calls and responses. We can then observe how a particular USB I/O device is interacting with its PC host.

Most of the code for DUET.SYS will be the same as NULLfilter.sys.

USB-related system calls are MajorFunctions with an IoControlCode of IOCTL_INTERNAL_USB_SUBMIT_URB. So, rather than passing down ALL of the MajorFunctions, we need to detect a USB-related call, display it and then optionally set up an InterceptReply routine so that we can display the response to the system call. The routine that checks the MajorFunction for a USB-related call and the InterceptReply routine is shown in Figure 11-21.

```
NTSTATUS CheckTheseFunctions(IN PDEVICE_OBJECT fido, IN PIRP Irp) {
	PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
	PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
	NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
	if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
	ULONG IoControlCode = stack->Parameters.DeviceIoControl.IoControlCode;
	bool Intercept = false; char Type = 0; USHORT USBfunction = 0; unsigned char Configuration;
	if (IoControlCode == IOCTL_INTERNAL_USB_SUBMIT_URB) {
		PURB urb = (PURB)stack->Parameters.Others.Argument1;
		USBfunction = urb->UrbHeader.Function;
		switch (USBfunction) {
			case URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE:
				Type = urb->UrbControlDescriptorRequest.DescriptorType;
				if (Type == 1) KdPrint((DRIVERNAME "Get Device Descriptor\n"));
				if (Type == 2) KdPrint((DRIVERNAME "Get Configuration Descriptor\n"));
				Intercept = true; break;
			case URB_FUNCTION_SELECT_CONFIGURATION:
				Configuration = urb->UrbSelectConfiguration.ConfigurationDescriptor->bConfigurationValue;
				KdPrint((DRIVERNAME " Set Configuration to %2.2x\n", Configuration)); break;
			case URB_FUNCTION_GET_DESCRIPTOR_FROM_ENDPOINT:
				KdPrint((DRIVERNAME "Get Report Descriptor\n")); Intercept = true; break;
			case URB_FUNCTION_CLASS_ENDPOINT:
				if (urb->UrbControlVendorClassRequest.Request == 10)
				KdPrint((DRIVERNAME "SET_IDLE = %4.4x\n", urb->UrbControlVendorClassRequest.Value));
				break;
			}
		}
	if (Intercept) {	// Need to intercept the reply so that the results can be displayed
		int Context = USBfunction | (Type << 16);  // InterceptReply needs the original Request
		IoCopyCurrentIrpStackLocationToNext(Irp);
		IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE)InterceptReply,
			(VOID*)Context, TRUE, TRUE, TRUE);
		return IoCallDriver(pdx->LowerDeviceObject, Irp); }
	else {
// Pass request down without additional processing
		IoSkipCurrentIrpStackLocation(Irp);
		status = IoCallDriver(pdx->LowerDeviceObject, Irp);
		IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
		return status; }
	}
```

```
NTSTATUS InterceptReply(IN PDEVICE_OBJECT fido, IN PIRP Irp, int Context) {
// The urb function gets changed to CONTROL_TRANSFER so recover the USBfunction and Type from original Request
    USHORT USBfunction = Context & 0x0ffff;
    USHORT Type = Context >> 16;
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    if (Irp->PendingReturned) IoMarkIrpPending(Irp);
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    PURB urb = (PURB)stack->Parameters.Others.Argument1;
    unsigned char* pByte = (unsigned char*)urb->UrbControlDescriptorRequest.TransferBuffer;
    unsigned long Length = urb->UrbControlDescriptorRequest.TransferBufferLength;

    KdPrint((DRIVERNAME "Response: Raw Data is %x (%dd) bytes = ", Length, Length));
    for (int i = 0; i<(int)Length; i++) {if ((i & 15) == 0) KdPrint(("\n")); KdPrint((" %2.2x", *pByte++));}
    KdPrint(("\n"));
    if (USBfunction == URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE) {
        pByte = (unsigned char*)urb->UrbControlDescriptorRequest.TransferBuffer;  // Reset data pointer
        DecodeDescriptor(pByte, Type); }
    KdPrint((DRIVERNAME "Intercept of %s completed\n", urbname[USBfunction]));
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return STATUS_SUCCESS;
    }
```

Figure 11-21. We detect USB functions with CheckTheseFunctions and hook to InterceptReply to look at their response if required

The DecodeDescriptor routine is a long lookup table and it is not shown here to save page count. It, along with the full source of all of the other routines, is included on the CDROM in a Visual C++ Workspace file.

TWO KEYBOARDS

This example takes DUET.SYS one stage further. Rather than just displaying the responses from the USB-related system calls, this example changes the information in the response before passing it back up to the calling routine. The calling routine does not know that this information was changed from the actual information on the real I/O device (the I/O device installer needs to know so that the filter program is correctly installed). This technique of changing the information "on-the-fly" can help you out of seemingly impossible situations but, as with all power tools, it should be used with respect and caution.

The dilemma I had was the attachment of two keyboards to a PC host – Windows integrated them both into its operating environment and I could enter characters on either keyboard. Impressive, but this was not what I wanted! My application needed to know which characters came from which keyboard and the operating system, when creating a single input data stream, discarded the data "source".

It was impossible to change the firmware of one of the keyboards since it was in ROM and was manufactured by a third party anyway. My solution was in two parts – a KBDfilter that created a "user" keyboard and my application that knows how to talk to a "user" keyboard.

We have learnt in previous chapters that an I/O device is defined via descriptors that it supplies to the PC host during enumeration. In the case of a keyboard, it is a HID device and its Report Descriptor describes its keyboard functionality. More specifically, the first two entries of a keyboards report descriptor will be Usage Page (Generic Desktop) and Usage (keyboard) – the operating system uses these two values to classify the I/O device as a system keyboard. To define a "user" keyboard we need only change the second entry to Usage (Vendor Defined) and the operating system will install it but won't classify it as a system input device. This modification is only one byte of the report descriptor and is changed by KBDfilter.sys as shown in Figure 11-22.

```
NTSTATUS InterceptReply(IN PDEVICE_OBJECT fido, IN PIRP Irp, IN PVOID Context) {
        PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
        if (Irp->PendingReturned) IoMarkIrpPending(Irp);
        PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
        PURB urb = (PURB)stack->Parameters.Others.Argument1;
        char* DescriptorBuffer = (char*)urb->UrbControlDescriptorRequest.TransferBuffer;

        DescriptorBuffer += 3;  // Point to the Usage
        if (*DescriptorBuffer == 6) *DescriptorBuffer = 0;        // Swap from "keyboard" to "Vendor defined"
        KdPrint(("Intercept completed\n"));
        IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
        return STATUS_SUCCESS;
        }
```

Figure 11-22. Changing the Report Descriptor at run-time

Refer to the TWOKBDs installation instructions on the CDROM for more detailed information. So now we have a “user” keyboard attached to the PC host. We now need to communicate with it.

I wrote a bare-bones, Visual C++ application based on the Win32app template to demonstrate the operation of two keyboards (a system keyboard and a user keyboard) called TwoKBDs.app.

This example program opens the user keyboard using standard HID methods and then waits for keystrokes. The ReadFile system call is implemented as a blocking function, which means that the function does not return until it has the data that was requested. To avoid “hanging” the other functions of the application program, I placed the ReadFile system call in its own program thread.

The example program collects characters from both keyboards and displays them on the user console in two different areas as shown in Figure 11-23. There is a lot of scope for improvement!!!!

<< Insert screenshot here >>

Figure 11-23. Collecting data from two keyboards

Using a Windows-supplied device driver.

The BlockIO.sys driver developed in the previous section solved the problem of moving large amounts of data between a PC host application and a BlockIO device. It has a simple interface and can be used to solve many design problems. There are occasions when a supplied Windows device driver is preferable – existing applications need to access a new USB I/O device for example. Microsoft provides a range of class drivers with Window 98 and Windows 2000 and the closest match to a simple bulk transfer class driver is the Mass Storage Device Driver.

The Mass Storage Device Driver treats your I/O device as if it were a floppy disk, hard drive or other storage device. An application program would access data on your I/O device the same way that it accesses files on a mass storage device i.e. using <drive><filename>
Our first stop is, as always, the descriptor tables. We set the class code to 8 to identify a Mass Storage device, set the bInterfaceSubClass to 6 to identify a transparent SCSI connection and set the bInterfaceProtocol to 50H to identify a BulkOnly transport implementation.

My example is an EZ-USB interface to a “hard disk” implemented using a Compact Flash card – the block diagram is shown in Figure 11-24 and the full schematic is included on the CDROM. The configuration descriptor is similar to the BlockIO example, needing two bulk endpoints in addition to the control endpoint.

Figure 11-24. A “hard-disk” implemented with FLASH memory

There is a lot of firmware to write for this example so I started with FrameWorks to implement the enumeration and then added modules to implement the mass-storage command extraction and implementation. The full source of these modules is included as a Keil project on the CDROM.

Cypress semiconductor has a more extensive version of this firmware that additionally supports IDE and SCSI hard drivers. Figure 11-25 shows a single chip interface using the EZ-FX microcontroller. The GPIF unit is used to gluelessly interface the ATAPI drive. This firmware source is available from Cypress for a no-fee license agreement. At the time of writing Cypress were starting the initial debug of an FX2 version of this design – the bus speed of 480Mb/s was delivering much faster apparent disk access times.

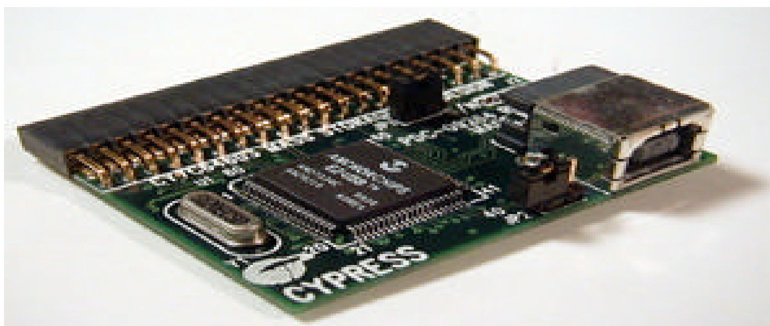

Figure 11-25. Glueless interface from EZ-FX to ATAPI

CHAPTER SUMMARY

USB was designed to move a lot of data very fast and this chapter has shown you how to accomplish this. We wrote a simple BlockIO device driver (simple is relative I know, it was MUCH harder than the HID example but is MUCH simpler than a NDIS network connection) that can be used as is or may be modified, with the help of Walter Oney's book, to meet your needs. We also wandered around the depths of the kernel world for a while and realized the power that it offers (and some of the dangers). It may be safer to force-fit your BlockIO application into a Mass Storage Driver; this is easier at the PC host side since Microsoft supply the driver but is more difficult at the I/O device since a range of pre-defined commands must be implemented. My example, or the one from Cypress Semiconductor, should get you up and running.

Once a driver is in place it is an easy for a PC host application program to transfer large blocks of data as it is to transfer single bytes using a HID interface.